Vrije Universiteit Amsterdam

Bachelor Thesis

Sketch-based distributed network monitoring

Author: Amin Soleimani (2640338)

1st supervisor: Lin Wang
2nd reader: Kees Verstoep

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 19, 2023

Abstract

Programmable switches offer the possibility of implementing network functions, such as DDoS detectors and firewalls, entirely in the data plane. These functions are commonly deployed across multiple switches within the network. The task of managing and coordinating the state across all instances of a network function is often delegated to a central controller in the control plane. However, this centralized approach often lacks scalability. In this thesis, we introduce three synchronization protocols for sketch-based network monitoring state synchronization entirely done in the data plane and analyze their performance. We implement these protocols on BMv2 Simple Switch architecture and test them with real-world network trace CAIDA. Our experimental results show that the distribution of state in sketch-based network monitoring is scalable and practical.

1 Introduction

The emergence of programmable data-plane switches in recent years has sparked interest in transitioning network functions (NFs) from dedicated hardware to these switches. Programmable switches have the capability of running NFs orders of magnitude faster than dedicated middleboxes and server-based implementations. Most NFs are stateful, and depending on the application, this state might be updated and read very frequently (e.g., per packet) [7,9]. These states are the information that an NF maintains to perform its intended task. Due to the distributed nature of network traffic, NFs need to be deployed across multiple switches in the network. This distribution is necessary as traffic can enter the network from various switches, and it is not practical to route all traffic through a single switch. Moreover, NFs often require to have a global view of the state to function properly. Control plane mechanisms are often employed to generate this global view. In this approach, a centralized controller periodically collects the state from switches, merges these states, and writes them back to switches. However, this method is inherently slow and not scalable, as NFs can process packets at Tbps rates, while controllers operate at much slower speeds. To overcome this limitation, the task of synchronizing the local state of switches should be done entirely in the data plane. The main research question of this thesis is: How to synchronize the local state of switches entirely in the data plane? In this thesis, we propose three synchronization protocols for performing network monitoring, inspired by the introduced framework in the SwiSh paper [7]. The main contributions of this thesis are the implementation of sketch-based network monitoring in the data plane switch, implementation of three protocols for synchronizing the local switch states entirely in the data plane, and experiments to compare the performance of these protocols. The remainder of this paper is structured as follows: section 2 describes the relevant background information. The system design is explained in section 3. In section 4 we discuss the implementation details. The system evaluation is performed in section 5. The final conclusions are discussed in section 6.

2 Background and Motivation

2.1 Programmable Switches

Software-Defined Networking (SDN) provides operators with the ability to manage their networks programmatically. In SDN, the control plane and data plane are distinct entities, with a single control plane overseeing multiple data plane devices. Essentially, the control plane dictates the network's behavior, while the data plane implements and executes this behavior on individual packets [10,12]. In programmable data plane switches, a multi-stage programmable pipeline, commonly known as Protocol Independent Switching Architecture (PISA), is employed for packet processing. The advantage of using a multi-stage pipeline instead of a single-stage processor is that forwarding a single packet often requires examining multiple header fields. Each stage of the pipeline can be programmed to analyze a different combination of fields [11,12,7]. PISA's multi-stage pipeline includes three main parts. The first is a Parser which parses the header fields from incoming packets. The next component consists of a series of match-action stages, each programmed to identify and potentially perform actions based on one or more matching header fields or metadata. Between stages, packets have the ability to convey extra information known as metadata, which is handled in the same way as packet header fields. Metadata can include various details such as the ingress port, transmit destination and queue, arrival timestamp, and data transferred from table to table without modifying the parsed representation of the packet, such as a virtual network identifier. The third component is the deparser, which re-serializes the packet that will be transmitted on the output link [7,10,11,12]. These three stages (parser, deparser, and match-action units) can be programmed using the P4 programming language [10].

2.2 Network Monitoring

Many network functions, such as traffic engineering (e.g., heavy hitter detection), security (e.g., DDoS detection), and anomaly detection (e.g., the entropy of source addresses) rely on network monitoring. Network monitoring can be categorized into active and passive approaches [1]. Active monitoring involves injecting the test traffic into the network and performing different types of measurements. On the other hand, passive monitoring entails capturing and analyzing the user-generated traffic at a specific point in the network. Flow monitoring is one of the common passive network monitoring approaches. RFC 7011 defines flow as “a set of packets or frames passing an Observation Point in the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties” [6]. Commonly, the 5-tuple attributes (source and destination IP, source and destination port, protocol) are used to identify different flows. Measuring flows with exact counting is not feasible due to the limited memory available on the network switches. One viable solution is to use sketches to store per-flow measurement data on the switch [2]. These sketches often provide a provable speed and accuracy guarantee [3]. Count-Min (CM) Sketch is a common sketch-based approach, used to estimate the frequency of elements in a dataset. CM sketch is a two-dimensional array of counters with width w and depth d. A separate pairwise-independent hash function is associated with each of the d rows in the sketch. The update procedure for an arrived item i comprises applying the corresponding hash function for each row j of the table to obtain the column index and incrementing its counter.

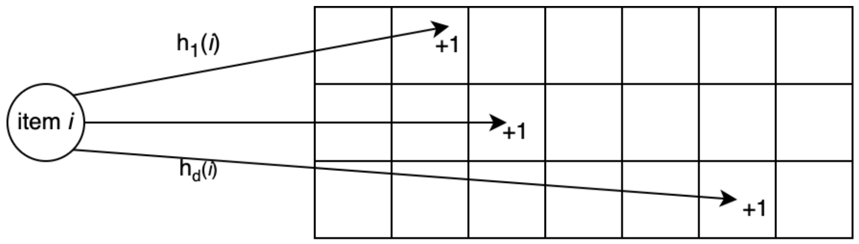

Figure 1: Each arriving item i is mapped to one column in each row of the array of counters. The counter in that position is then incremented.

The query procedure is similar to updates. For each row j, we obtain the column index by applying the associated hash function and reading the counter’s value. The estimated count is given by taking the least value read from the table for a given item [4,5]. Another desirable property of sketch data structures is mergeability. Assuming two CM sketches having the same size and dimensionality, and using the same hash functions for each row, we can merge them by summing the corresponding counters in each row [5]. This makes CM sketch a suitable option to use in distributed settings.

Running Nfs on Programmable Switches. Data centers employ a diverse array of network functions (NFs), encompassing both standard features such as NAT, firewalls, and load balancers, as well as specialized proprietary functions tailored to their unique requirements. These NFs, commonly run on commodity servers, require substantial hardware resources due to their crucial role in processing a significant portion of incoming data center traffic, leading to high expenses [7]. The emergence of programmable data-plane switches in recent years has enabled more efficient approaches for implementing these NFs. these switches are capable of running these functions at line-rate, which is orders of magnitude faster than the current server-based implementations [7,8].

Distributed Deployment. Due to the presence of multiple entry points/paths in data center networks, the distributed deployment of NFs across multiple switches is necessary to effectively capture traffic across different routes, while also being crucial for scalability and switch failure tolerance. As mentioned in the previous section, many NFs rely on network monitoring. With the distributed deployment of these NFs, the local state (e.g., the flow count) on each switch needs to be shared with other switches [7,9].

The Need for Data-Plane Replication. The switch state replication is commonly done through the control-plane. A central controller periodically retrieves the state from each switch, processes the information, and updates the switches with the merged state. Reading the switch state register from the control-plane is orders of magnitude slower than reading it from the data plane. In SwiSh, the authors recorded an average latency of 507 milliseconds when reading a sketch from the on-switch control-plane. The authors managed to read the same sketch in 486 microseconds from the data-plane [7]. These results indicate that control-plane replication approaches are not scalable.

3 System Design

In this project, we introduce three state sharing protocols for network monitoring purposes, each addressing a different consistency requirement. Switches in the network maintain a replica of the shared state through the exchange of replication packets, which are Ethernet packets. The fields within these packets vary depending on the protocols being employed. We use the terms replication packets and update packets interchangeably. Every switch in the network is assigned a distinct switch ID, serving as a means to distinguish incoming update packets from other switches. For this project, we assume switches never fail and the replication/acknowledgment packets are never dropped. However, we will provide a recovery mechanism for each protocol.

3.1 Synchronous Synchronization Protocol

This protocol ensures strong consistency by employing blocking writes to the shared state variable (sketch). However, due to the limited memory capacity of the switches, the packets themselves need to be buffered in the controller during the synchronization process, which limits the throughput.

Bookkeeping. Each switch implementing this protocol retains a local sketch to track its own local updates, along with an identical sketch for each switch in the network. Switches also employ a counter to keep track of the number of incoming packets, serving as the sequence number in the generated update packets. Moreover, to detect duplication/loss of update packets, switches keep track of the sequence number of the most recent updates received from each individual switch.

Update Packet Format. In addition to the Ethernet header, each update packet encompasses the switch ID, sequence number, and the indices (one for each row in the sketch) specifying which columns should be incremented within the sketch associated with the corresponding switch ID of the sender. Furthermore, each packet includes a single bit indicating whether the packet is an update or an acknowledgment (ACK).

3.1.1 Packet Processing and Synchronization

The packet processing and synchronization procedures can be classified into three distinct steps.

Receiving New Packet. Upon the arrival of a new packet from external sources (not from another switch), the switch initiates the process by incrementing its packet counter. Following this, for each row of the sketch, the switch applies the corresponding hash function to the 5-tuple associated with the packet. This operation yields the column indices where the counters will be incremented when the synchronization is performed. To leverage the faster hashing capability of the switch hardware compared to the controller, this step is performed in the data plane [13]. Subsequently, the switch sends the packet, along with its assigned sequence number, to the controller for buffering. Simultaneously, the switch generates update packets utilizing the obtained column indices from the previous step. These update packets are then transmitted to other switches within the network. It is important to note that this approach is only viable under the assumption that ACKs are not dropped. In scenarios where there is a possibility of dropped ACKs, the update packets should be generated by the controller, and sent only after all ACKs for the previous update are received.

Receiving Update Packet. When a switch receives an update packet, it first verifies if the sequence number of the update packet matches the expected sequence number for the sending switch. If the sequence number aligns with the expected value, the switch proceeds to increment the counter to signify the sequence number for the subsequent expected update from that switch. Furthermore, it increments the indices within the sketch associated with the switch ID of the sender, according to the information provided in the update packet. Lastly, the switch overwrites the switch ID field in the update packet with its ID and sets the designated bit to indicate that the packet is an ACK. This modified packet is then transmitted back to the original sender.

Receiving ACK. Received ACKs are directly transmitted to the controller. Once the controller has received ACKs from all other switches within the network, it proceeds to send the buffered packet back to the switch for forwarding. Finally, the switch updates its own local sketch and proceeds to forward the packet as intended.

3.1.2 Handling Packet Duplication and Reordering

The update packets include a sequence number, enabling switches to apply updates in the correct order and to reject duplicate updates. To mitigate the impact of dropped packets, the controller incorporates a timer for each buffered packet. Upon timer expiration, the controller retransmits the update packet to switches that have not yet acknowledged the update.

3.2 Asynchronous Synchronization Protocol

In the synchronous protocol, the act of writing to the shared state variable carries a substantial cost. To address this issue, this protocol implements asynchronous writes to the shared state. This means that when a switch receives a packet, it modifies its local state, emits any corresponding output packet without any delays, and subsequently sends the update to other switches. As a consequence, this protocol only provides strong eventual consistency. To prevent excessive network traffic caused by a high volume of update packets, switches have the capability to accumulate multiple writes before transmitting the replication packets to other switches.

Bookkeeping. Switches running this protocol maintain a local sketch to monitor their own local updates, as well as an identical sketch for every switch in the network. To support batching, switches employ a counter to track the number of incoming packets.

Update Packet Format. Update packets include the switch ID, the index of the sketch column, and the corresponding values. Since each register array can only be accessed once per packet, one update packet is generated per sketch column. The number of values contained in each update packet corresponds to the depth of the sketch.

3.2.1 Packet Processing and Synchronization

As previously mentioned, incoming packets are counted locally, and the corresponding output packet is promptly emitted. When the packet counter reaches the defined batch size limit, the local sketch is transmitted to the other switches in the network. Since reading the full sketch during the processing of a single packet is not possible, the sketch is sent column by column.

Merging Update Packet. Each column in the sketch is a Grow-only CRDT counter [14]. Merging the received updates into the sketch associated with the sender can be done by comparing the value in the update packet with the corresponding value in the sender's sketch column at the specified index. The maximum value is then selected as the merged result.

3.2.2 Handling Packet Duplication and Reordering

G-Counters exhibit the desirable properties of being commutative and idempotent. These properties imply that packet duplication and reordering have no impact on the final result of the updates. Additionally, periodic synchronization resolves the issue of dropped packets. However, to guarantee the eventual consistency property, a mechanism needs to be employed to periodically send update packets to other switches, independent of the arrival of any new packets. This mechanism helps mitigate the impact of lost update packets, as the periodic transmission ensures that updates are consistently propagated throughout the network over time.

3.3 Window-Based Asynchronous Synchronization Protocol

The asynchronous protocol mitigates the issues of high latency and low throughput encountered in the synchronous model by relaxing the level of consistency it offers. This can lead to inconsistent replicas, as the eventual consistency does not provide any guarantees regarding the convergence time. The window-based synchronization model provides r-relaxed strong linearizability. This implies that each query of the shared state is permitted to overlook a bounded number of preceding updates.

Bookkeeping. Each switch implementing this protocol retains a window ID counter, a local sketch to track its own local updates, and an identical sketch per switch in the network. Additionally, two sketches named *sync* and *merge* are utilized to facilitate window synchronization. Moreover, all switches maintain two bitmaps: one to track ACKs indicating that other switches have received its updates, and the other to indicate if it has received all updates from other switches in the network. In this protocol, switches send an ACK only after the full sketch is received. To achieve this, switches keep a counter per switch in the network, tracking the index of the next column that will be updated.

Update Packet Format. Update packets include the switch ID, window ID, a single bit indicating whether the packet is an update or an ACK, the index of the sketch column, and the corresponding values. Similar to the previous protocol, the transmission of the sketch occurs column by column.

3.3.1 Packet Processing and Synchronization

In this protocol, updates are accumulated into windows, and the window advancement is synchronized between switches. The fundamental assumption in this approach is that all switches within the network advance their window at the same time. At any given time, writes resulting from incoming external packets are applied to the local sketch of the switch, while the content of the *merge* sketch is identical (synchronized) across all switches.

Synchronization. During each synchronization event, the content of the local sketch is copied to the *sync* sketch, and the local sketch is reset to zero. Subsequently, the *sync* sketch is transmitted to other switches within the network. Switches store the received updates in the assigned sketch of the respective senders. Once the full sketch is received, the bitmap is updated and an ACK is sent back.

Window Advancement. Reception of all updates and ACKs from other switches indicates the end of the synchronization round. Subsequently, the sketches containing the updates from other switches, as well as the sync sketch on each switch are merged into the merge sketch. The merging process involves adding the corresponding columns from these sketches together. Lastly, the bitmaps are set back to zero, and the window ID counter is incremented.

3.3.2 Handling Packet Duplication and Reordering

Replication packets include the window ID, which enables switches to discard updates that do not correspond to the current window. Furthermore, by keeping track of the next expected column index for each switch, the protocol enables the detection of duplicated and dropped packets. During window synchronization, switches can set a timer. When the timer expires, any updates that have not been acknowledged by other switches will be retransmitted.

4 Implementation

For this project, we provide an implementation for the introduced protocols for the bmv2 simple_switch_grpc. This implementation is available at `https://github.com/amin91s/p4`. It's important to note that software switches have certain limitations when compared to hardware switches. In this section, we will discuss these limitations and propose potential solutions to overcome them. We then delve into the implementation details of the protocols.

4.1 Software Switch Limitations

Timers. Time-triggered events are currently not supported in the software switches. To overcome this limitation, we adopt an approach where a host, connected to all switches, takes on the responsibility of periodically transmitting a custom timer packet to the switches. This packet serves as a synchronization signal, enabling switches to coordinate their window advancement.

Packet Generation. Unlike many hardware switches, software switches can not generate new packets. As previously mentioned, the register array can only be accessed once per packet. This means that operations on the entire sketch are performed index by index, utilizing the packet generators. To address this limitation, we leverage the “clone-preserving-field-list” and “recirculate-preserving-field-list” extern functions. These functions enable us to clone and recirculate packets while preserving specific field lists. Additionally, we take advantage of the simple switch’s ability to remove and add new headers to the incoming packets and emit them accordingly. For instance, in the asynchronous protocol, the incoming packet that triggers the synchronization protocol is cloned from the ingress to the egress pipeline, with a specific metadata bit set to indicate this packet should be recirculated back to the ingress. Prior to the recirculation of this packet for the first round, the TCP and IP headers are set to invalid, effectively removing them. Then the update header is set to valid, and the packet is truncated to the size of the Ethernet header plus the update header. The same bit in the metadata is then checked to identify the packet back in the ingress block. Besides this bit, the metadata field also includes a register that is used to track the recirculation round (i.e., the column index in the sketch). In each round, the round register is incremented, and the packet is cloned to the egress pipeline to be recirculated again. The values from the local sketch are then copied to the packet itself (which was cloned in the previous round). This packet is then multicasted to other switches in the network. *Timer* packets in the asynchronous model are similarly utilized to support window synchronization procedures.

4.2 Implementation Details

For our implementation, it is crucial that switches can identify their own switch ID. Since it is not possible to hardcode this ID in the p4 code, a special packet containing a unique ID is initially sent to every switch in the network. Additionally, a match action table can be used to map switch IDs to the corresponding sketch.

Synchronous Protocol. In this protocol, controllers establish a gRPC connection to switches for the P4Runtime service. Packets between the switch and controller are transmitted to the CPU port. Packet IO messages between the switch and controller are exchanged using the bi-directional stream. We define the packet IO header in the P4 program. Packets sent to the controller are packet-in and packets from the controller are packet-out packets. These headers can be used to include additional information.

Window-Based Asynchronous Protocol. For this project, we assume that replication and acknowledgment packets are always reliably delivered within the network. This assumption allows us to only use a single timer to signify the start of the synchronization round. In our implementation, timer signals are ignored if the previous sync round is not completed yet (this never happens due to the assumption). To relax this assumption, two other timers with relatively shorter intervals can be utilized to monitor the received ACKs and updates, retransmitting them is necessary.

5 Evaluation

The protocols are implemented on BMv2 Simple Switch architecture [16]. The experiments are done on a virtual machine running Ubuntu 20.04, with 4 allocated CPU cores, 8 GB of ram, and running Mininet version 2.3.1. The test topology consists of three switches in a fully connected network. A host is connected to each switch, which is used to send the packets to the switch. The controller in the synchronous model is written in python3. Furthermore, we have constructed two packet generators designed to send traffic to the switch. The first generator replays packet traces sourced from CAIDA [17], while the second one generates random TCP/IP packets. Both generators utilize raw sockets in Python to send the packets.

5.1 Synchronization Time

We use the switch logs to evaluate the synchronization protocols. Alongside the automatically generated logs, BMv2 allows user-defined messages to be logged as well. In our measurements, we log a specific message right before generating the first update packet, signaling the start of synchronization. In the synchronous model, the message that signifies the end of synchronization is logged immediately after the buffered packet is transmitted out of the switch. For the window-based protocol, the completion of synchronization is when the window ID register is incremented. For the asynchronous model, the message is logged on the receiver of the update immediately after the last entry in the sketch is merged. In the synchronous protocol, each received packet only updates one sketch index. Consequently, there are no other parameters to manipulate when measuring the synchronization time. On the other hand, in asynchronous and window-based models, the entire sketch is transmitted during the synchronization process. The synchronization for the synchronous model took on average 13 ms.

From the data in Table 1, we can see that the window-based model takes on average 1.6 times longer to complete the synchronization. The observed result aligns with our expectations, as the synchronization process in the window-based model is inherently more complex than in the asynchronous model. In the window-based model, the procedure involves reading and writing six sketches, and waiting for the reception of all ACKs before advancing the window, while in the asynchronous model, only two registers are operated on.

Table 1: Synchronization times for Asynchronous and window-based models (in ms)

Number of Register Entries	Asynchronous	Window-based
8	16	34
16	32	56
32	66	99
64	103	146
128	179	289

Table 2: Number of packets sent for different batch sizes

Packets Per Second	Batch Size
$\approx$250	1
$\approx$1666	10
$\approx$5000	30
$\approx$11111	50
$\approx$20000	100

5.2 Batching Updates in the Asynchronous Model

For this experiment, we disabled the switch logging capability. We noticed that this optimization allowed us to send packets 5 times faster. As mentioned in Section 3, switches can accumulate writes together, and send updates in batches. The result for measuring performance for different batch sizes can be seen in Table 2. In this experiment, each register contains 16 entries.

Table 2 illustrates that increasing the batch size leads to improved performance. However, it is essential to note that this improvement comes at the cost of a longer time required for the state to converge. The optimal batch size ultimately depends on the NF's tolerance for inconsistent state. We can also observe that the packet rate for batch size 1 is lower than the rate for the synchronous model, which is roughly 500 packets per second (with logging disabled). This is due to the transmission of the whole sketch in the asynchronous model.

6 Conclusions

This thesis introduced three protocols to support the synchronization of the local switch state entirely in the data plane. Each protocol guarantees a different level of state consistency. The synchronous protocol prioritizes strong consistency but at the expense of performance. In contrast, the asynchronous model, while delivering a good performance, offers only eventual consistency. On the other hand, the window-based approach strikes a balance by providing a bounded time within which the state is converged. We also implemented sketch-based network monitoring to overcome the memory limitation on the programmable switches. Based on the experiments conducted on the BMv2 Simple Switch architecture, the obtained results indicate that the window-based protocol is a practical option for NFs that can tolerate some level of inconsistency in the shared state. These findings suggest that the window-based approach provides a reasonable trade-off between performance and state convergence, making it a viable and attractive choice for such NFs.

References

[1] R. Hofstede et al., "Flow Monitoring explained: From packet capture to data analysis with NetFlow and IPFIX," IEEE Communications Surveys & Tutorials, vol. 16, no. 4, pp. 2037–2064, 2014. doi:10.1109/comst.2014.2321898

[2] S.-Y. Kim, C. Jung, R. Jang, Aziz Mohaisen, and DaeHun Nyang, "Count-Less: A Counting Sketch for the Data Plane of High-Speed Switches," Nov. 2021, doi: `https://doi.org/10.48550/arxiv.2111.02759`.

[3] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," Journal of Algorithms, vol. 55, no. 1, pp. 58–75, Apr. 2005, doi: `https://doi.org/10.1016/j.jalgor.2003.12.001`.

[4] G. Cormode, "Count-Min Sketch." Accessed: Jul. 19, 2023. [Online]. Available: `http://dimacs.rutgers.edu/~graham/pubs/papers/cmencyc.pdf`

[5] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," Journal of Algorithms, vol. 55, no. 1, pp. 58–75, Apr. 2005, doi: `https://doi.org/10.1016/j.jalgor.2003.12.001`.

[6] P. Aitken, B. Claise, and B. Trammell, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," IETF, Sep. 01, 2013. `https://www.rfc-editor.org/rfc/rfc7011.html` (accessed Jul. 19, 2023).

[7] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. "Swish: Distributed shared state abstractions for programmable switches." In USENIX NSDI 2022. USENIX, April 2022.

[8] Zeno, Lior and Ports, Dan R. K. and Nelson, Jacob and Silberstein, Mark. "SwiShmem: Distributed Shared State Abstractions for Programmable Switches." In Proceedings of the 19th ACM Workshop on Hot Topics in Networks, HotNets '20, page 160–167, New York, NY, USA, 2020. Association for Computing Machinery.

[9] P. Bosshart et al., "P4," ACM SIGCOMM Computer Communication Review, vol. 44, no. 3, pp. 87–95, Jul. 2014, doi: https://doi.org/10.1145/2656877.2656890.

[10] L. Peterson, C. Cascone, and B. Davie, Software-Defined Networks. 2021.

[11] I. Butun, Y. K. Tuncel, and K. Oztoprak, "Application Layer Packet Processing Using PISA Switches," Sensors, vol. 21, no. 23, p. 8010, Nov. 2021, doi: https://doi.org/10.3390/s21238010.

[12] S. Yoo and X. Chen, "Secure Keyed Hashing on Programmable Switches," Aug. 2021, doi: https://doi.org/10.1145/3472873.3472881.

[13] Nuno Preguiça, C. Baquero, and M. Shapiro, "Conflict-free Replicated Data Types (CRDTs)," arXiv (Cornell University), May 2018, https://doi.org/10.1007/978-3-319-63962-8/_185-1.

[14] J. Xing, W. Wu, and A. Chen, "Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries," www.usenix.org, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/xing (accessed Jul. 19, 2023).

[15] "BEHAVIORAL MODEL (bmv2)," GitHub, Jul. 18, 2023. https://github.com/p4lang/behavioral-model/tree/main (accessed Jul. 19, 2023).

[16] CAIDA. "The CAIDA UCSD Anonymized Internet Traces - 2019." https://www.caida.org/catalog/datasets/passive_dataset